1. 上课约定须知

课程主题：Flink 源码解析 -- 第三次课
上课时间：20:00 - 23:00
课件休息：21:30 左右 休息10分钟
课前签到：如果能听见音乐，能看到画面，请在直播间扣 666 签到

2. 上次内容总结

第一次课主要讲解的是：Flink RPC 和 Standalone 集群启动中主节点 JobManager 启动源码剖析，主要包含以下三方面：

```
1、Flink RPC 详解
2、Flink 集群启动脚本分析
3、Flink 主节点 JobManager 启动分析
```

在 JobManager 启动过程中，主要做了四件大事：

```
1、initializeServices(....) 初始化各种服务
2、WebMonitorEndpoint 启动
3、ResourceManager 启动
4、Dispatcher 启动
```

第二次课主要讲解的是 Flink 集群启动的 从节点启动流程的源码剖析，主要涉及到的知识点：

```
1、Flink TaskManager 启动源码分析
2、TaskManagerServices 初始化源码剖析
3、TaskExecutor 启动和源码分析
4、TaskExecutor 注册和心跳源码剖析
```

在这个 TaskManager 启动过程中，最重要的事情，就是在 TaskManager 初始化了一些基础服务和一些对外提供服务的核心服务之后就启动 TaskExecutor，向 JobManager(ResourceManager) 进行 TaskManager 的注册，并且在注册成功之后，维持 TaskManager(TaskExecutor) 和 JobManager(ResourceManager) 的心跳。

3. 上次作业复盘

使用 Flink RPC 组件模拟实现 YARN，这个需求和我之前在讲 Spark 源码的时候，讲解的使用 Akka 模拟实现 YARN 的需求是类似的，只不过需要使用的技术是 Flink RPC 组件！

实现要求：

```
1、资源集群主节点叫做： ResourceManager，负责管理整个集群的资源
2、资源集群从节点叫做： TaskExecutor，负责提供资源
3、ResourceManager 启动的时候，要启动一个验活服务，制定一种机制（比如：某个 TaskExecutor 的连续5次心跳未接收到，则认为该节点死亡）实现下线处理
4、TaskExecutor 启动之后，需要向 ResourceManager 注册，待注册成功之后，执行资源（按照 Slot 进行抽象）汇报 和 维持跟主节点 ResourceManager 之间的心跳以便 ResourceManager 识别到 TaskExecutor 的存活状态
```

4. 本次内容大纲

今天的课程是 Flink 源码剖析 的第三次：主要内容是 Flink 应用程序的提交。主要包含 Job 的三层 Graph 处理（首先根据应用程序构建 StreamGraph，然后构建 JobGraph，并行化之后生成 ExecutionGraph）。

```
1、Flink 编程套路总结
2、Flink 提交执行脚本分析
3、Flink CliFrontend 提交应用程序源码剖析
4、ExecutionEnvironment 源码解析
5、Job 提交流程源码分析
6、WebMonitorEndpoint 处理 RestClient 的 JobSubmit 请求
7、StreamGraph 构建和提交源码解析
8、JobGragh 构建和提交源码解析
9、ExecutionGragh 构建和提交源码解析
```

到此为止，接下来要做的事情，就是：

```
1、JobMaster 像 ResourceManager 申请 slot 资源
2、执行 Task
```

5. 详细课堂内容

5.1. Flink Program 编程套路总结

Flink：高级的抽象：针对一个任意类型数据的任意类型计算逻辑的任务复杂和数据规模的计算应用程序编写条路的抽象！

Flink 底层提供了一个功能完善且复杂的分布式流式计算引擎，但是上层的应用 API 却很简单，简单来说，把整个 Flink 应用程序的编写，抽象成三个方面：

- 执行环境 ExectionEnvironment
- 数据抽象 DataSet DataStream
- 逻辑操作 Source Transformation Sink

所以 Flink 的应用程序在编写的时候，基本是一个简单的统一套路：

```
1、获取执行环境对象
   StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
2、通过执行环境对象，注册数据源Source，得到数据抽象
   DataStream ds = env.socketTextStream(...)
3、调用数据抽象的各种Transformation执行逻辑计算
   DataStream resultDS = ds.flatMap(...).keyBy(...).sum(...);
4、将各种Transformation执行完毕之后得到的计算结果数据抽象注册 Sink
   resultDS.addSink(...)
5、提交Job执行
   env.execute(...)
```

基本路数，和 Spark 一致，并且，在 Flink-1.12 版本中， DataStream 已经具备高效批处理操作处理了。更加做到了流批处理的统一（API统一）。据 "Flink Forward Asia 2020 在线峰会" 阿里流式计算负责人王峰介绍：在 Flink-1.13 版本，将会完全统一流批处理的 API。

在 Flink 应用程序中，其实所有的操作，都是 StreamOperator，分为 SourceOperator，StreamOperator，SinkOperator，然后能被优化的 Operator 就会 chain 在一起，形成一个 OperatorChain。

理解这么几个概念：转换（x => y）= 函数 = Function => Transformation => StreamOperator => OperatorChain（并行化之后，得到 StreamTask 执行）

5.2. Flink Job 提交脚本解析

当编写好 Flink 的应用程序，正常的提交方式为：打成 jar 包，通过 flink 命令来进行提交。

flink 命令脚本的底层，是通过 java 命令启动：CliFrontend 类来启动 JVM 进程执行任务的构造和提交。

```
flink run xxx.jar class arg1 arg2
```

具体可以参考官网：https://ci.apache.org/projects/flink/flink-docs-stable/deployment/cli.html

阅读 flink shell 脚本的内容可知道，最终会转到：CliFrontend 来执行提交处理。

5.3. CliFrontend 提交分析

当用户把 Flink 应用程序打成 jar 使用 flink run ... 的 shell 命令提交的时候，底层是通过 CliFrontend 来处理。底层的逻辑，就是通过反射来调用用户程序的 main() 方法执行。

在刚组建内部，主要有以下几件事要做：

```
1、根据 flink 后面的执行命令来确定执行方法(run ==> run(params))
2、解析 main 参数，构建 PackagedProgram，然后执行 PackagedProgram
3、通过反射获取应用程序的 main 方法的实例，通过反射调用执行起来
```

总的来说：就是准备执行 Program 所需要的配置，jar包，运行主类等的必要的信息，然后提交执行（通过反射，转而执行到用户自定义应用程序的 main 方法）。

假如以例子程序： WordCount 来说：转到执行 WordCount 的 main() 方法！

5.4. ExecutionEnvironment 源码解析

Flink 应用程序的执行，首先就是创建运行环境 StreamExecutionEnvironment，一般在企业环境中，都是通过 getExecutionEnvironment() 来获取 ExecutionEnvironment，如果是本地运行的话，则会获取到：LocalStreamEnvironment，如果是提交到 Flink 集群运行，则获取到：StreamExecutionEnvironment。

```
final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
```

StreamExecutionEnvironment 是 Flink 应用程序的执行入口，提供了一些重要的操作机制：

```
1、提供了 readTextFile(), socketTextStream(), createInput(), addSource() 等方法去对接数据源

2、提供了 setParallelism() 设置应用程序的并行度

3、StreamExecutionEnvironment 管理了 ExecutionConfig 对象，该对象负责 Job 执行的一些行为配置管理。还管理了 Configuration 管理一些其他的配置

4、StreamExecutionEnvironment 管理了一个 List<Transformation<?>> transformations 成员变量，该成员变量，主要用于保存 Job 的各种算子转化得到的 Transformation，把这些 Transformation 按照逻辑拼接起来，就能得到 StreamGragh（Transformation -> StreamOperator -> StreamNode）

5、StreamExecutionEnvironment 提供了 execute() 方法主要用于提交 Job 执行。该方法接收的参数就是：StreamGraph
```

StreamExecutionEnvironment 是 Flink 应用程序执行的上下文，提供了很多功能，不过重点关注以上五点即可。

5.5. Job 提交流程源码分析

核心流程如下：

```
// 核心入口
env.execute("Streaming WordCount");

    // 负责生成 StreamGraph
    // 负责执行 StreamGraph
    execute(getStreamGraph(jobName));
```

第一步：getStreamGraph(jobName) 生成 StreamGraph 解析

```
// 入口
StreamGraph streamGraph = getStreamGraph(jobName, true);

    // 通过 StreamGraphGenerator 来生成 StreamGraph
    StreamGraph streamGraph =
getStreamGraphGenerator().setJobName(jobName).generate();

        streamGraph = new StreamGraph(....)
        for(Transformation<?> transformation : transformations) {
            transform(transformation);
        }
```

transform(transformation) 的内部实现：

```
transform(transformation);

    // 先递归处理该 Transformation 的输入
    Collection<Integer> inputIds = transform(transform.getInput());

    // 将 Transformation 变成 Operator 设置到 StreamGraph 中，其实就是添加 StreamNode
    streamGraph.addOperator(....)

    // 设置该 StreamNode 的并行度
    streamGraph.setParallelism(transform.getId(), parallelism);

    // 设置该 StreamNode 的入边 SreamEdge
    for(Integer inputId : inputIds) {
        streamGraph.addEdge(inputId, transform.getId(), 0);

            // 内部实现
            // 构建 StreamNode 之间的 边（StreamEdge） 对象
            StreamEdge edge = new StreamEdge(upstreamNode, downstreamNode, ...)
            // TODO_MA 注释： 给 上游 StreamNode 设置 出边
            getStreamNode(edge.getSourceId()).addOutEdge(edge);
            // TODO_MA 注释： 给 下游 StreamNode 设置 入边
            getStreamNode(edge.getTargetId()).addInEdge(edge);
    }
```

第二步：execute(StreamGraph) 解析

```
// 入口
JobClient jobClient = executeAsync(streamGraph);

    // 执行一个 SreamGraph
    executorFactory.getExecutor(configuration).execute(streamGraph,
configuration);

        // 第一件事：由 StreamGraph 生成 JobGragh
        JobGraph jobGraph = PipelineExecutorUtils.getJobGraph(pipeline,
configuration);

        // 第二件事：通过 RestClusterClient 提交 JobGraph 到Flink集群
        clusterClient.submitJob(jobGraph)
```

继续提交：

```
// 通过 RestClusterClient 来提交 JobGraph
RestClusterClient.submitJob(JobGraph jobGraph);

    // 继续提交
    RestClusterClient.sendRetriableRequest()

        // 通过 RestClient 提交
        RestClient.sendRequest(webMonitorHost, webMonitorPort, ...)

            // 继续提交

RestClient.submitRequest(targetAddress,targetPort,httpRequest,responseType)
```

最终通过 channel 把请求数据，发给 WebMonitorEndpoint 中的 JobSubmitHandler 来执行处理。

关于客户端的 Job 的提交，总结为这么几个步骤：

```
1、用户根据 Flink 应用程序的编写套路，写好应用程序，打成jar包，通过 flink run 的命令来执行提交
2、这个命令的底层，其实是执行： CliFrontend 组件来执行提交
3、这个 CliFrontend 的内部，会通过反射的技术，来转交执行到 用户自定义应用程序的 main()
4、先获取 StreamExecutionEnvironment
5、执行算子：其实就是从 算子 --- function --- StreamOperator --- Transformation
6、执行 StreamExecutionEnvironment 的 executor 方法来执行提交
7、首先遍历 StreamExecutionEnvironment 的 transformations 这个list 来生成 StrewamGraph
8、具体的内部的提交是通过 RestClusterClient 来执行提交
9、在通过 RestClusterClient 提交之前，其实还会做一件事：把 SreamGraph 变成 JobGraph，也还会先把 JobGraph 持久化成为一个磁盘文件
10、在这个 RestClusterClient 的内部，其实是通过 RestClient 来提交
11、RestClient 其实在初始化的时候，就初始化了一个 Netty 客户端
12、通过 封装一个 HttpRequest 对象，通过 Netty 客户端 链接服务端，发送请求对象
```

5.6. WebMonitorEndpoint 处理 RestClient 的 JobSubmit 请求

最终处理这个请求： JobSubmitHandler 来处理！

核心入口：

```
// JobManager 服务端处理入口
JobSubmitHandler.handleRequest();

    // 恢复得到 JobGraph
    JobGraph jobGraph = loadJobGraph(requestBody, nameToFile);

    // 通过 Dispatcher 提交 JobGraph
    Dispatcher.submitJob(jobGraph, timeout)
```

Dispatcher 的提交执行逻辑：

```
Dispatcher.persistAndRunJob()

    // 保存 JobGraph 在 ZK 上
    jobGraphWriter.putJobGraph(jobGraph);

    // 提交 JobGraph 执行
    // Dispatcher 最重要的事情 就是帮忙启动 JobMaster
    Dispatcher.runJob(jobGraph);

        // 第一件事：主要的事情，是创建 JobMaster
        // 在这句代码的内部，有一件非常重要的事情： 把 JobGraph 变成 ExecutionGraph
        // 看起来是创建 JobManagerRunner，其实是得道： JobManagerRunnerImpl
        Dispatcher.createJobManagerRunner(jobGraph);
            // 初始化 new JobManagerRunnerImpl
            new JobManagerRunnerImpl()
                // 初始化 JobMaster
                new JobMaster(...)
                    // 创建 DefaultScheduler
                    this.schedulerNG =
createScheduler(jobManagerJobMetricGroup);
                        schedulerNGFactory.createInstance(...)
                            new DefaultScheduler()
                                super()

this.executionGraph=createAndRestoreExecutionGraph()
                                        // 重点：创建 ExecutionGraph
                                        newExecutionGraph =
createExecutionGraph()
                    // 然后跳转到 JobMaster 的 onStart() 方法。
                    onStart()

        // 第二件事：启动 JobMaster
        Dispatcher.startJobManagerRunner()
            jobManagerRunner.start();
                leaderElectionService.start(this);
                    ZooKeeperLeaderElectionService.isLeader()

JobManagerRunnerImpl.grantLeadership(issuedLeaderSessionID);

verifyJobSchedulingStatusAndStartJobManager(leaderSessionID);
                                startJobMaster(leaderSessionId);
```

注意两对概念：主从架构

```
1、ResourceManager + TaskExecutor
2、JobMaster（Driver） + StreamTask（Executor）
```

主节点启动的时候的三大角色的功能作用：

```
1、WebMonitorEndpoint    启动了一个 Netty 服务端接收 客户端的 rest 请求，然后根据 请求 url 来调用对应的 handler来执行处理
2、ResourceManager    只是负责管理集群的资源，维持和从节点之间的心跳
3、Dispatcher    WebMonitorEndpoint当中的 JobSubmitHandler 接收到一个请求处理之后恢复得到一个 JobGraph 转交给 Dispatcher 去生成和启动一个 JobMaster 用来启动这个 Job
```

最后的结论： 启动的这个 JobMaster 负责这个 Job 中的所有的 Task 的 slot 的申请和 任务的派发，状态的跟踪，容错，还有checkpoint等各种操作！

接上面的继续：startJobMaster() 继续执行：最重要是两件事：

```
1、JobMaster 向 ResourceManager 注册，注册成功之后，维持心跳！
2、JobMaster 向 ResourceManager 申请 Slot，部署 StreamTask 运行！
```

```
startJobMaster(leaderSessionId);

    runningJobsRegistry.setJobRunning(jobGraph.getJobID());
    jobMasterService.start(new JobMasterId(leaderSessionId));

        // 内部主要做两件事：
        // startJobMasterServices();
        // resetAndStartScheduler();
        startJobExecution(newJobMasterId), RpcUtils.INF_TIMEOUT);

            // 第一件事：跑起来 JobMaster 相关的服务，主要是注册 和 心跳
            startJobMasterServices();

                startHeartbeatServices();
                slotPool.start(getFencingToken(), getAddress(),
getMainThreadExecutor());
                scheduler.start(getMainThreadExecutor());
                reconnectToResourceManager(new FlinkException("..."));

                    closeResourceManagerConnection(cause);
                    tryConnectToResourceManager();

                        connectToResourceManager();

                            resourceManagerConnection = new
ResourceManagerConnection()
                            resourceManagerConnection.start();

                                createNewRegistration();
                                newRegistration.startRegistration();

                                    register(...)
                                        invokeRegistration(...);
                                            gateway.registerJobManager(...)
                                                // 完成注册
                                                registerJobMasterInternal()
```

```
                // 监听 ResourceManager
                resourceManagerLeadRetriever.start(new
ResourceManagerLeaderListener());

            // 第二件事：开始申请 Slot，并且部署 Task
            resetAndStartScheduler();

                JobMaster.startScheduling()

                    schedulerNG.startScheduling();

                        // 启动所有的服务协调组
                        startAllOperatorCoordinators();
                        // 开始调度
                        startSchedulingInternal();

                            // 更改状态
                            prepareExecutionGraphForNgScheduling();
                            // 开始调度
                            schedulingStrategy.startScheduling();

                                allocateSlotsAndDeploy(...)
```

接着继续：

```
allocateSlotsAndDeploy(...)

    DefaultScheduler.allocateSlotsAndDeploy(executionVertexDeploymentOptions);

        // 申请 Slot
        allocateSlots(executionVertexDeploymentOptions);

        // 部署 Task 运行
        waitForAllSlotsAndDeploy(deploymentHandles);
```

5.7. Flink Graph 演变

Flink 的一个 Job，最终，归根结底，还是构建一个高效率的能用于分布式并行执行的 DAG 执行图。

FLink 的三层图概念：

```
1、StreamGraph    就是通过用户编写程序时指定的算子进行逻辑拼接的
        简单说：就是进行算子拼接
2、JobGraph   其实就是在StreamGraph的基础之上做了一定的优化，然后生成的逻辑执行图
        简单说：就是把能   优化拼接在一起，放在一个Task中执行的算子的整合和优化 chain 在一起形
成 OperatorChain，这个操作就类似于 Spark Stage 切分
3、ExecutionGraph   再把JobGraph进行并行化生成ExecutionGraph
        简单说：其实ExecutionGraph就是JobGraph的并行化版本。
```

其实，最后还有物理执行图（当一个 Flink Stream Job 中的所有的 Task都被调度执行起来了之后的状态）！

1、StreamGraph转变成JobGraph： 帮我们把上下游两个相邻算子如果能chain到一起，则chain到一起做优化
2、JobGraph转变成ExecutionGraph： chain到一起的多个Operator就会组成一个OperatorChain，当OperatorChain执行的时候，到底要执行多少个 Task，则就需要把 DAG 进行并行化变成实实在在的Task来调度执行

最开始：
    dataStream.xx1().xxx2().xxx3().....xxxn();
    evn.execute();

到最后：
    List<StreamTask> 执行（不同的StreamTask(StreamTask内部逻辑计算操作不一样)）

总结要点
    相邻两个阶段之间的StreamTask是有关系的。（到底哪些上游StreamTask生产数据给下游消费StreamTask）
    Shuffle关系!

一个 Flink 流式作业，从 Client 提交到 Flink 集群，到最后执行，总共会经历四种不同的状态。总的来说：

1、Client 首先根据用户编写的代码生成 StreamGraph，然后把 StreamGraph 构建成 JobGraph 提交给 Flink 集群主节点
2、然后启动的 JobMaster 在接收到 JobGraph 后，会对其进行并行化生成 ExecutionGraph 后调度启动 StreamTask 执行。
3、StreamTask 并行化的运行在 Flink 集群中的，就是最终的物理执行图状态结构。

Flink 中的执行图可以分成四层：**StreamGraph** ==> **JobGraph** ==> **ExecutionGraph** ==> **物理执行图。**

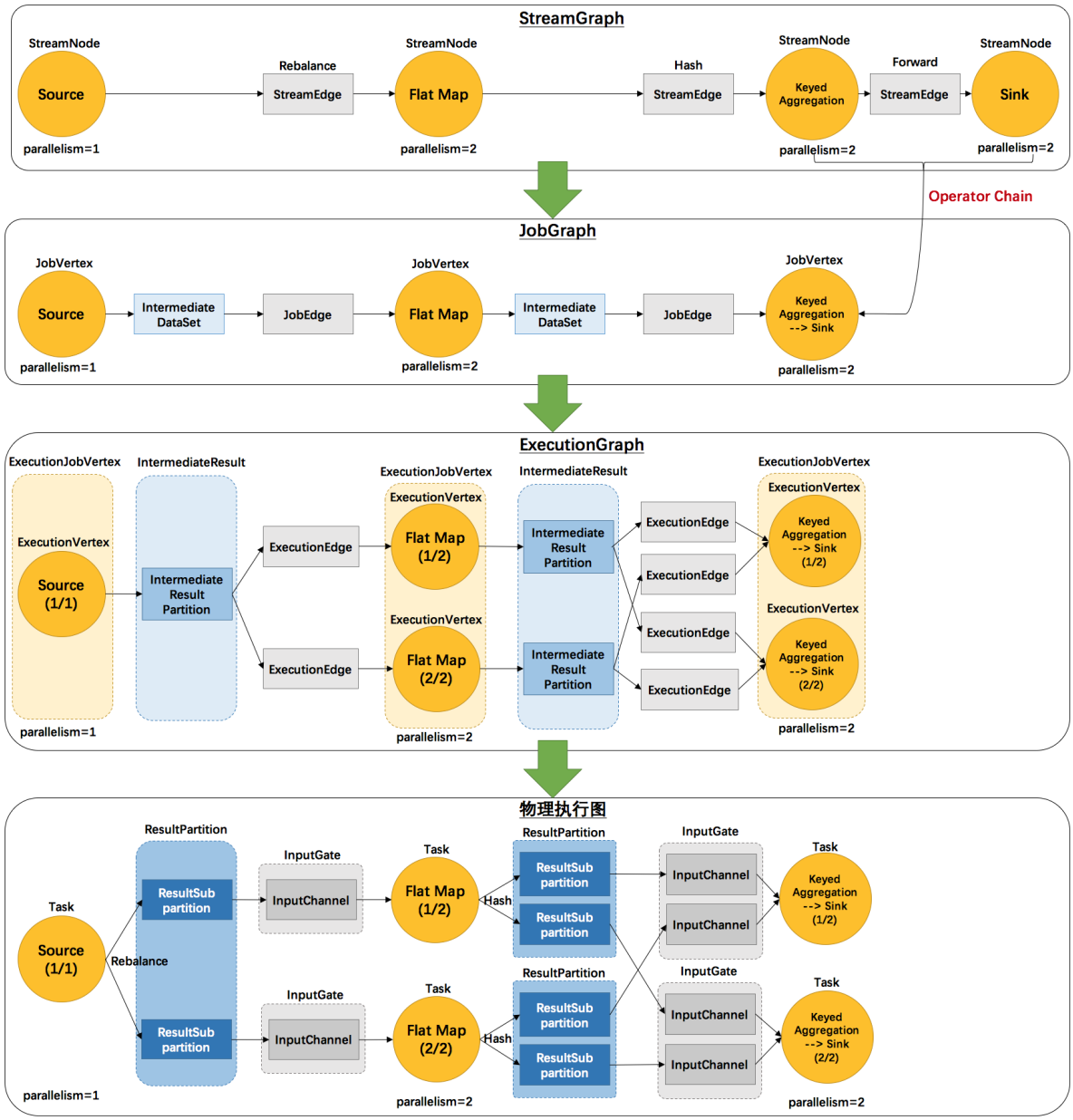
StreamGraph：是根据用户通过 Stream API 编写的代码生成的最初的图。用来表示程序的拓扑结构。

JobGraph：StreamGraph 经过优化后生成了 JobGraph，提交给 JobManager 的数据结构。主要的优化为，将多个符合条件的节点 chain 在一起作为一个节点，这样可以减少数据在节点之间流动所需要的序列化反序列化传输消耗。

ExecutionGraph：JobManager 根据 JobGraph 生成 ExecutionGraph。ExecutionGraph 是 JobGraph 的并行化版本，是调度层最核心的数据结构。

物理执行图：JobManager 根据 ExecutionGraph 对 Job 进行调度后，在各个 TaskManager 上部署 Task 后形成的图，并不是一个具体的数据结构。

关于这四层之间的演变，请看下图：



上面这张图清晰的给出了 Flink 各个图的工作原理和转换过程。其中最后一个物理执行图并非 Flink 的数据结构，而是程序开始执行后，各个 Task 分布在不同的节点上，所形成的物理上的关系表示：

- 从 JobGraph 的图里可以看到，数据从上一个 operator（JobVertex） 流到下一个 operator（JobVertex） 的过程中，上游作为生产者提供了 IntermediateDataSet，而下游作为消费者需要 JobEdge。事实上，JobEdge 是一个通信管道，连接了上游生产的 dataset和下游的 JobVertex 节点。
- 在 JobGraph 转换到 ExecutionGraph 的过程中，主要发生了以下转变：
  - 加入了并行度的概念，成为真正可调度的图结构
  - 生成了与 JobVertex 对应的 ExecutionJobVertex，ExecutionVertex，与 IntermediateDataSet 对应的 IntermediateResult 和IntermediateResultPartition 等，并行将通过这些类实现
- ExecutionGraph 已经可以用于调度任务。我们可以看到，Flink 根据该图生成了一一对应的 Task，每个 Task 对应一个ExecutionGraph 的一个 Execution。Task 用 InputGate、InputChannel 和 ResultPartition 对应了上面图中的 IntermediateResult和 ExecutionEdge。

那么，设计中为什么要设计这么四层执行逻辑呢？它的意义是什么？

```
1、StreamGraph 是对用户逻辑的映射
2、JobGraph 在 StreamGraph 基础上进行了一些优化，比如把一部分操作串成 chain 以提高效率
3、ExecutionGraph 是为了调度存在的，加入了并行处理的概念
4、物理执行结构：真正执行的是 Task 及其相关结构
```

5.7.1. StreamGraph 构建和提交源码解析

StreamGraph：根据用户通过 Stream API 编写的代码生成的最初的图。Flink 把每一个算子 transform 成一个对流的转换（比如 SingleOutputStreamOperator， 它就是一个 DataStream 的子类），并且注册到执行环境中，用于生成 StreamGraph。

它包含的主要抽象概念有：

```
1、StreamNode：用来代表 operator 的类，并具有所有相关的属性，如并发度、入边和出边等。
2、StreamEdge：表示连接两个 StreamNode 的边。
```

源码核心代码入口：

```
StreamExecutionEnvironment.execute(getStreamGraph(jobName));
```

其中：

```
1、getStreamGraph(jobName) 先生成一个 SteamGraphGenerator，然后调用 generate() 方法生成一个 StreamGraph
2、StreamExecutionEnvironment.execute() 用于执行一个 StreamGraph
```

StreamGraph 生成过程中，生成 StreamNode 的代码入口：

```
streamGraph.addOperator(vertexID, slotSharingGroup, coLocationGroup,
operatorFactory, inTypeInfo, outTypeInfo, operatorName);
```

StreamGraph 生成过程中，生成 StreamEdge 的代码入口：

```
streamGraph.addEdge(inputId, transform.getId(), 0);
```

5.7.2. JobGraph 构建和提交源码解析

JobGraph：StreamGraph 经过优化后生成了 JobGraph，提交给 JobManager 的数据结构

它包含的主要抽象概念有：

1、JobVertex：经过优化后符合条件的多个 StreamNode 可能会 chain 在一起生成一个 JobVertex，即一个JobVertex 包含一个或多个 operator，JobVertex 的输入是 JobEdge，输出是 IntermediateDataSet。

2、IntermediateDataSet：表示 JobVertex 的输出，即经过 operator 处理产生的数据集。producer 是JobVertex，consumer 是 JobEdge。

3、JobEdge：代表了job graph中的一条数据传输通道。source 是 IntermediateDataSet，target 是 JobVertex。即数据通过JobEdge由IntermediateDataSet传递给目标JobVertex。

源码核心代码入口：

```
final JobGraph jobGraph = PipelineExecutorUtils.getJobGraph(pipeline,
configuration);
```

经过层层递进：

```
StreamingJobGraphGenerator.createJobGraph(this, jobID);
```

在 StreamGraph 构建 JobGragh 的过程中，最重要的事情就是 operator 的 chain 优化，那么到底什么样的情况的下 Operator 能chain 在一起呢?

```
// 1、下游节点的入度为1 （也就是说下游节点没有来自其他节点的输入）
downStreamVertex.getInEdges().size() == 1;

// 2、上下游节点都在同一个 slot group 中
upStreamVertex.isSameSlotSharingGroup(downStreamVertex);

// 3、前后算子不为空
!(downStreamOperator == null || upStreamOperator == null);

// 4、上游节点的 chain 策略为 ALWAYS 或 HEAD（只能与下游链接，不能与上游链接，Source 默认是 HEAD）
!upStreamOperator.getChainingStrategy() == ChainingStrategy.NEVER;

// 5、下游节点的 chain 策略为 ALWAYS（可以与上下游链接，map、flatmap、filter 等默认是 ALWAYS）
!downStreamOperator.getChainingStrategy() != ChainingStrategy.ALWAYS;

// 6、两个节点间物理分区逻辑是 ForwardPartitioner
(edge.getPartitioner() instanceof ForwardPartitioner);

// 7、两个算子间的shuffle方式不等于批处理模式
edge.getShuffleMode() != ShuffleMode.BATCH;

// 8、上下游的并行度一致
upStreamVertex.getParallelism() == downStreamVertex.getParallelism();

// 9、用户没有禁用 chain
streamGraph.isChainingEnabled();
```

5.7.3. ExecutionGraph 构建和提交源码解析

ExecutionGraph：JobManager(JobMaster) 根据 JobGraph 生成 ExecutionGraph。ExecutionGraph 是JobGraph 的并行化版本，是调度层最核心的数据结构。

它包含的主要抽象概念有：

1、ExecutionJobVertex：和JobGraph中的JobVertex一一对应。每一个ExecutionJobVertex都有和并发度一样多的 ExecutionVertex。

2、ExecutionVertex：表示ExecutionJobVertex的其中一个并发子任务，输入是ExecutionEdge，输出是IntermediateResultPartition。

3、IntermediateResult：和JobGraph中的IntermediateDataSet一一对应。一个IntermediateResult包含多个IntermediateResultPartition，其个数等于该operator的并发度。

4、IntermediateResultPartition：表示ExecutionVertex的一个输出分区，producer是ExecutionVertex，consumer是若干个ExecutionEdge。

5、ExecutionEdge：表示ExecutionVertex的输入，source是IntermediateResultPartition，target是ExecutionVertex。source和target都只能是一个。

6、Execution：是执行一个 ExecutionVertex 的一次尝试。当发生故障或者数据需要重算的情况下 ExecutionVertex 可能会有多个 ExecutionAttemptID。一个 Execution 通过 ExecutionAttemptID 来唯一标识。JM和TM之间关于 task 的部署和 task status 的更新都是通过 ExecutionAttemptID 来确定消息接受者。

源码核心代码入口：

```
ExecutionGraph executioinGraph = SchedulerBase.createAndRestoreExecutionGraph()
```

在 SchedulerBase 这个类的内部，有两个成员变量：一个是 JobGraph，一个是 ExecutioinGraph

在创建 SchedulerBase 的子类：DefaultScheduler 的实例对象的时候，会在 SchedulerBase 的构造方法中去生成 ExecutionGraph。

源码核心流程：

```
SchedulerBase.createAndRestoreExecutionGraph()

    ExecutionGraph newExecutionGraph = createExecutionGraph(...)

        ExecutionGraphBuilder.buildGraph(jobGraph, ....)

            // 创建 ExecutionGraph 对象
            executionGraph = (prior != null) ? prior : new ExecutionGraph(...)
            // 生成 JobGraph 的 JSON 表达形式

executionGraph.setJsonPlan(JsonPlanGenerator.generatePlan(jobGraph));
            // 重点，从 JobGraph 构建 ExecutionGraph
            executionGraph.attachJobGraph(sortedTopology);

                // 遍历 JobVertex 执行并行化生成 ExecutioinVertex
                for(JobVertex jobVertex : topologiallySorted) {

                    // 每一个 JobVertex 对应到一个 ExecutionJobVertex
                    ExecutionJobVertex ejv = new ExecutionJobVertex(jobGraph,
jobVertex);
                    ejv.connectToPredecessors(this.intermediateResults);
```

```
                        List<JobEdge> inputs = jobVertex.getInputs();
                        for(int num = 0; num < inputs.size(); num++) {

                            JobEdge edge = inputs.get(num);
                            IntermediateResult ires =
intermediateDataSets.get(edgeID);
                            this.inputs.add(ires);

                            // 根据并行度来设置 ExecutionVertex
                            for(int i = 0; i < parallelism; i++) {
                                ExecutionVertex ev = taskVertices[i];
                                ev.connectSource(num, ires, edge,
consumerIndex);
                            }
                        }
                }
```

5.7.4. 物理执行图

物理执行图：JobManager 根据 ExecutionGraph 对 Job 进行调度后，在各个TaskManager 上部署 Task 后形成的“图”，并不是一个具体的数据结构。

它包含的主要抽象概念有：

```
1、Task：Execution被调度后在分配的 TaskManager 中启动对应的 Task。Task 包裹了具有用户执行逻辑的 operator。

2、ResultPartition：代表由一个Task的生成的数据，和ExecutionGraph中的IntermediateResultPartition一一对应。

3、ResultSubpartition：是ResultPartition的一个子分区。每个ResultPartition包含多个ResultSubpartition，其数目要由下游消费 Task 数和 DistributionPattern 来决定。

4、InputGate：代表Task的输入封装，和JobGraph中JobEdge一一对应。每个InputGate消费了一个或多个的ResultPartition。

5、InputChannel：每个InputGate会包含一个以上的InputChannel，和ExecutionGraph中的ExecutionEdge一一对应，也和ResultSubpartition一对一地相连，即一个InputChannel接收一个ResultSubpartition的输出。
```

6. 本次课程总结

本次课程，主要讲解的是，Flink 客户端如何把一个 Job 提交到集群上去运行。主要涉及 Graph 的处理，和 Job 的提交过程的分析。

```
1、关于 Job 的提交过程分析！
2、关于 Job 的三层图构建和分析！
```

下次课的主要内容，是 Task 的 Slot 申请和 执行源码分析！

7. 次课程作业

使用 Flink RPC 组件模拟实现 YARN，这个需求和我之前在讲 Spark 源码的时候，讲解的使用 Akka 模拟实现 YARN 的需求是类似的，只不过需要使用的技术是 Flink RPC 组件！

实现要求：

```
1、资源集群主节点叫做： ResourceManager，负责管理整个集群的资源
2、资源集群从节点叫做： TaskExecutor，负责提供资源
3、ResourceManager 启动的时候，要启动一个验活服务，制定一种机制（比如：某个 TaskExecutor 的连续5次心跳未接收到，则认为该节点死亡）实现下线处理
4、TaskExecutor 启动之后，需要向 ResourceManager 注册，待注册成功之后，执行资源（按照 Slot 进行抽象）汇报 和 维持跟主节点 ResourceManager 之间的心跳以便 ResourceManager 识别到 TaskExecutor 的存活状态
```